Automating Software Feature Verification

Gerard J. Holzmann and Margaret H. Smith

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A significant part of the call processing software for Lucent's new PathStar™ access server [FSW98] was checked with formal verification techniques. The verification system we built for this purpose, named FeaVer is accessed via a standard web browser. The system maintains a database of feature requirements together with the results of the most recently performed verifications. Via the browser the user can invoke new verification runs, which are performed in the background with the help of a logic model checking tool. Requirement violations are reported either as high-level message sequence charts or as detailed source-level execution traces of the system source. A main strength of the system is its capability to detect potential feature interaction problems at an early stage of systems design, the type of problem that is difficult to detect with traditional testing techniques.

Error reports are typically generated by the system within minutes after a comprehensive check is initiated, allowing near interactive probing of feature requirements and quick confirmation (or rejection) of the validity of tentative software fixes.

1. Introduction

Distributed systems software can be difficult to test. The detailed behavior of such a system typically depends on subtle timings of events and on the relative speed of execution of concurrent processes. This means that errors, once they manifest themselves, can be very hard to reproduce, and it means that when the system passes a series of tests, one cannot safely conclude that the same tests can never fail. The situation is further complicated when we deal with increasingly complex software with multiple feature packages, a phenomenon that most PC users today will be familiar with. The problem also exists in the systems code of large telephone switching systems.

The best-known manifestation of the problem in call-processing applications is the so-called *feature interaction problem* [KK98]. Telephone companies can compete with elaborate feature packages that are offered to customers, ranging from standard features such as call forwarding, to more obscure variations like call parking. The number of distinct features offered on a main switch today can be well over one hundred. Each of these features can require a different response to the same basic set of events that can occur on a subscriber line, and thus the feature interaction problem is born. With just 25 features there can already be 2^{25} possible feature combinations. If each combination could be tested in a second, it would take about a year to test all combinations. By any standard, this is an undesirable strategy. For a simple example of feature interaction, consider what the required response of the switch is if a customer has both an anonymous call rejection and a call forwarding feature enabled simultaneously. Which of these two features should take precedence when an anonymous call arrives for this customer?

Fortunately, in practice the situation is not quite this bad. Telcordia, the former BellCore, has issued standards on feature behavior that switch providers must comply with [B92-96]. According to these standards, some feature combinations are not allowed, some are non-conflicting, and for some a feature precedence relation is prescribed that determines which feature behavior is to take precedence in case of conflict. (For the example above, for instance, the rules state that the anonymous call rejection feature should take precedence.) Unfortunately, the rules from the standards are not always complete, and they are sometimes hard

to interpret unambiguously. The task of systematically checking if the Telcordia standards have been implemented correctly by a vendor of a switching system therefore remains formidable.

Methods that can be used to mechanically verify distributed systems software should be of considerable value in industrial software design. Specifically, we are interested here in methods that can be used to formally verify the call processing software, and specifically the feature code, for a new commercial switch. We will describe a system named FeaVer, that can accomplish this feat.

Earlier attempts to apply automated verification techniques to distributed software applications generally have relied on hand crafted formal models, often produced by verification experts over a period of months in collaboration with the developers of an application, e.g. [CAB98][S98]. Because of the time required to construct formal models by hand, detailed changes in the source application cannot easily be tracked without a significant reinvestment of time and energy. By eliminating the need for hand crafted models, the system we will describe can be used to verify virtually every version of an application, tracking the evolving source throughout the design cycle.

In the next few sections we will discuss the central components of the FeaVer feature verification system:

- Mechanized model extraction: a method for mechanically extracting verification models from implementation level code, controlled by a user-defined conversion table.
- Formulating properties: defining the set of formal requirements that the application has to satisfy. In our case many properties could be derived from the Telcordia standards for call processing feature implementation. Others define more specific local requirements or are more exploratory in nature. The use of a database of correctness properties is comparable to the use of test suites and test objectives in a traditional testing method.
- Logic model checking: the method that is used to mechanically verify if the system satisfies or possibly can violate one or more of the stated requirements.
- System support: the mechanics of the verification process, including the use of a system of networked PCs, called TrailBlazer, to execute verification jobs in parallel.

We conclude the paper with a summary of our findings.

2. Mechanized model extraction

It is known that it is not possible to devise an algorithm that could prove arbitrary properties of arbitrary C or C++ programs. It is not even possible to mechanically prove a single specific, property such as program termination for arbitrary programs [T36][S65]. So if we want to be able to render proofs, we have no choice but to restrict ourselves to a smaller class of programs. An example of such a class is the set of all finite state programs: programs that on any given input can generate only a finite number of possible program states (i.e., memory configurations) when executed. We call a simplified program of this type a *model*. The set of all possible executions for a finite state model defines a finite directed, and possibly cyclic, graph. Even without explicitly constructing the complete graph, which can still be large, we can now reason about feasible and infeasible paths in the graph, and prove if certain executions are possible or not. This is precisely what a logic model checker is designed to do.

The first problem to be solved then is to reduce a given C or C++ program to a meaningful finite state model that can be analyzed. The reduction will bring a loss of information, so it has to be chosen in such a way that relevant information is preserved and irrelevant detail removed. What is 'relevant' and what is not depends on the properties that we are interested in proving about the program. If, for instance, the functioning of the billing subsystem is not mentioned in any of the system requirements we check, than all access to and manipulation of billing data can be stripped from the program to produce the model. Some care has to be taken, though, to guarantee that the removal of code preserves our ability to find all property violations. The following procedure will ensure this.

All assignments and function calls that have been tagged as irrelevant to the verification effort are replaced with a *skip* (a dummy no-op in the modeling language). All conditional choices that refer to data objects tagged as irrelevant are replaced by nondeterministic choices. The use of nondeterminism is a standard

reduction technique that can be used to make a model more general, broadening its scope. The nondeterminism tells the model checker that all possible outcomes of a choice should be considered equally possible, not just one specifically computed choice. The original computation of the system is preserved as one of the possible abstracted computations, and the scope of the verification is therefore not restricted. If no property violation exists in the reduced system, we can safely conclude that no property violation can exist in the original application

The reduction method is 'fail-safe' in the sense that if we chose the reduction incorrectly, the above result still holds true, although the reverse does not [AL91],[CGL94],[K95],[B99]. It is possible, for instance, that the full expansion of an error trace for a property violation detected in the reduced system does not correspond to a valid execution of the original application. If this happens it constitutes a proof that information was inadvertently stripped from the system that was in fact relevant to the verification. In this case at least one of the conditional choices in the abstract trace will turn out to be invalid in the concrete trace, not matching assignments to data objects earlier in the trace. These data objects are now known to be relevant to be properties being verified, and the reduction can be adjusted accordingly. Typically a few iterations of this type suffice to converge on a stable definition of an abstraction that can be used to extract a verifiable model from a program text, as we will discuss in more detail below.

The PathStar Code

In the verification of the code for the PathStar access server [FSW98], shown in Figure 1, our focus is exclusively on the verification of telephony features. Since we are not looking for faults in the sequential code of device drivers, process schedulers, memory allocation routines, billing subsystems, etc., the function of such code can be abstracted. For device drivers, for instance, it means that the abstractions used do not enable us to check that a device driver administers dialtone correctly when given the appropriate command by the controller, but it *does* enable us to check that the controller can only issue the appropriate commands when required, and cannot fail to do so [HS99].

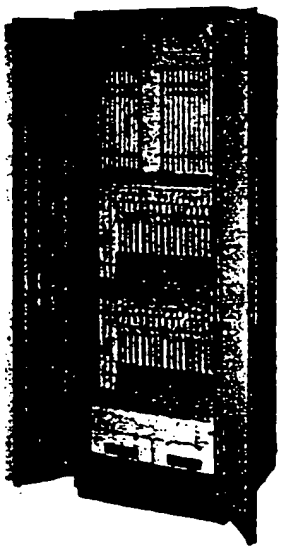

Fig. 1 — The PathStar™ Access Server
Providing data and voice service over a variety of media.

In the PathStar code the function of the controller is specified in a large routine that defines the central state machine for all basic call processing and feature behavior. This routine, roughly 1,600 lines of C source, is executed concurrently by a varying number of processes, jointly responsible for the handling of incoming and outgoing calls. In the extracted model for this code, we carefully preserve all concurrent behavior and the complete execution of the state machine, be it in slightly abstracted form.

Nondeterministic test drivers in the model are used to generalize the behavior of all parts of the system that

are external to the state machine: subscriber behavior, connected devices, remote switches. The source text of the original program is preserved in the abstract model so that we easily reproduce a concrete trace from any abstract error trace that is discovered.

With the reduction process we have outlined, the control flow of the original source is preserved in the reduced model. Data access, however, passes through a user-defined abstraction filter. This filter, defined as a conversion lookup table, determines which operations are irrelevant to the properties to be verified (e.g., function calls for billing and accounting) and which need to be represented either literally, with an equivalent representation in the language of the model checker, or in more abstract form. The irrelevant operations are mapped into the null operation of the model checker.

The Conversion Table

From all different types of statements that appear in the PathStar call processing code, about 60% are mapped to an equivalent statement in the extracted model (i.e., they are preserved in the abstraction), cf. Figure 2. This includes all statements that cause messages to be sent from one process to another (like call requests, call progress and call termination signals), and all statements that are used to record or test the call state of a subscriber.

The remaining statements and conditionals are abstracted in one of three ways, depending on their relevance to the verification effort.

1. A statement that is entirely outside the scope of the verification is replaced with *skip* and thereby stripped from the model, as also discussed above. This applies to about 30% of the cases.
2. If a statement is partially relevant, the conversion table defines a mapping function that preserves only the relevant part and suppresses the rest. For example, we do not use the absolute values of timers in our verifications. For the properties we define it suffices to know only if a timer is running or not, and therefore the integer range of possible timer values can be reduced with a mapping function to the boolean values *true* and *false*: indicating whether or not a timer might expire. This mapping could not be used if we were to include requirements on the real-time performance of the PathStar switch. Other examples of this type of abstraction are cases where the details of an operation are irrelevant, but the possible outcomes are not. For instance, digit analysis can be an involved operation that is mostly irrelevant to the functional correctness of the call processing code. Only relevant is that the controller deals correctly with the possible outcomes of this operation: to respond properly when an abbreviated number or a feature access code is recognized, to start routing the call if it is determined that sufficient digits were collected, or to wait for the subscriber to provide more digits, with the proper timers set to guard the inter-digit timing interval. In this case the conversion table replaces the operation with a nondeterministic choice of the possible outcomes.
3. The third type of abstraction is used when an operation is fully relevant and needs to be preserved in the model, with only syntax adjustments.

Method	*Percentage of Code*
Fully abstracted (stripped)	*30%*
Functional and Non-deterministic abstraction (mapped)	*10%*
Not abstracted (preserved)	*60%*

Fig. 2 — Ratio of basic types of abstractions applied

To track changes in the source text, and to retain the capability to extract models, we only have to keep the conversion table up to date, rather than a fully detailed hand-crafted model. Some changes in the source require no update at all. This is the case, for instance, if code is copied or moved without the introduction of new types of data manipulation. When a new type of data access appears, the model extractor warns the user and prompts for a new entry in the conversion table. In most cases the new entry can be defined without knowing anything about the purpose of the change or it's impact on behavior. Typically, a week's worth of upgrades of the call processing code translates into ten minutes of work on a revision of the conversion table before a fully mechanized verification of all properties can be repeated. More detail on the definition of conversion tables can be found in [HS99].

Assumpti ns about the Environment

The call processing code in PathStar interacts with a number of entities in its environment: subscribers, remote switches, database servers, and the like. The task of constructing precisely detailed behavior definitions for each of these entities would be both formidable and redundant [H97].

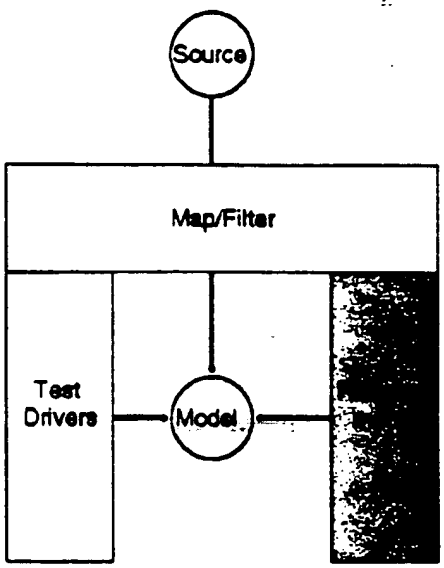

Fig. 3 — Verification Context
The verification context consists of a conversion map that defines the level of abstraction, test drivers that capture the essential assumptions about the environment, and a database of properties that define the system requirements.

For each remote entity that interacts with the call processing controller it suffices to construct a small abstract model that captures a conservative estimate of the possible behaviors of these entities in a general way. Note that our objective is not to verify the correct behavior of the remote entities, but that of our own switch *despite* the presence of possibly ill-behaved remote entities. The system requirements, test drivers, and the conversion map together define the verification context, as illustrated in Figure 3.

We can, for instance, define an abstract model for generic subscriber behavior with a simple demon that can nondeterministically select an action from all the possible actions that a subscriber might take at each point in a call: going on- or off-hook, flashing the hook, dialing feature access codes, etc. Similarly we can model the possible responses from a remote switch to call requests from the local controller, using a demon that can generate possible responses nondeterministically.

Abstractions such as these, based on nondeterminism, achieve two objectives: they remove complexity by removing extraneous detail, while at the same time broadening the scope of the verification by representing larger classes of possible behavior, instead of selected instances of specific behavior.

3. Formulating Properties

The database for feature verification of the PathStar code that we have constructed contains approximately 80 properties for 20 features. For each feature set the database further defines one or more provisioning constraints. When verifying the correct implementation of any given feature we must, for instance, be able to specify that the feature is to be enabled, and that incompatible features are to remain disabled. These additional constraints could be included in the definitions of the properties themselves, but the extra information would hamper their readability. By decoupling provisioning detail from functional properties we can more easily experiment with different types of provisionings on a common set of properties. It is, for instance, possible to check the correct implementation of feature precedence relations by deliberately enabling and disabling higher precedence features. In the absence of an explicit provisioning constraint, the

model checker will assume no knowledge about the enabledness or disabledness of features, leaving this to nondeterministic choice.

For each feature, each property is verified for each related provisioning constraint. For 80 properties this translates, in our current database, into approximately 200 separate verification runs. The number of runs to be performed for a full verification of the source code can change with the addition or deletion of properties or provisioning constraints.

An Example

As a simple example we will consider the formalization of the requirement that when a non-ringing phone is picked up, dial-tone is generated. Clearly, there are some exceptions to this rule. When the line is provisioned as a hotline, with a *direct call* feature, or when the line is provisioned with the *denial of originating service* 'feature' then no dialtone will be generated. To check this property, therefore, we need to define a provisioning constraint that disables such higher priority features.

One method to specify this property is to use a simple form of linear temporal logic. Temporal logic, introduced in the late seventies for the concise formulation of correctness properties of concurrent systems [P77], defines a small number of operators that allow us to reason about executions. In temporal logic the example property can be specified as follows:

$$\Box\ (\ \text{offhook} \rightarrow X \Diamond\ (\ \text{dialtone} \vee \text{onhook})\ )$$

In this case we allow for the possibility that the subscriber returns the phone onhook before actually hearing the dialtone, which would of course be valid. The truth value of a temporal formula is evaluated over execution sequences. This means that if we evaluate the formula at any given point in a system's execution it would return true if and only if the complete remainder of the execution from that point forward satisfies the property stated.

Three unary temporal operators are used in the formula above: $\Box$ (always), X (next), and $\Diamond$ (eventually). $\Box$ p states that p is true now and will remain invariantly true throughout the rest of the computation. X p states that p will be true after the next execution step. $\Diamond$ p states that p is either true now or it will become true within a finite number of future execution steps. The right-arrow, $\rightarrow$, denotes logical implication: $(p \rightarrow q)$ means $(\neg p \vee q)$, where $\neg$ is logical negation and $\vee$ is logical or.

The model checker will use this formula to check if there can be any system executions that would violate the property. This procedure works by first negating the formula, so that we get a formalization of a violating execution. The negated formula for the example can also be derived manually as follows, using standard rewrite rules from boolean and temporal logic:

$$\begin{aligned}
&\neg \Box\ (\ \text{offhook} \rightarrow\ X \Diamond\ (\ \text{dialtone} \vee \text{onhook})\ ) &&\equiv\\
&\Diamond \neg\ (\ \text{offhook} \rightarrow\ X \Diamond\ (\ \text{dialtone} \vee \text{onhook})\ ) &&\equiv\\
&\Diamond \neg\ (\ \neg \text{offhook} \vee X \Diamond\ (\ \text{dialtone} \vee \text{onhook})\ ) &&\equiv\\
&\Diamond\ (\ \text{offhook} \wedge \neg\ \ X \Diamond\ (\ \text{dialtone} \vee \text{onhook})\ ) &&\equiv\\
&\Diamond\ (\ \text{offhook} \wedge\ \ X \neg \Diamond\ (\ \text{dialtone} \vee \text{onhook})\ ) &&\equiv\\
&\Diamond\ (\ \text{offhook} \wedge\ \ X \Box\ (\neg\ \text{dialtone} \wedge \neg\ \text{onhook})\ )
\end{aligned}$$

This negated formula can be converted mechanically [GPVW95] into a 2-state ω-automaton, illustrated in Figure 4, which is used in the model checking process.

Timeline Editor

For the specification of complex behavior, e.g., to capture properties on the correct functioning of a six-way conference call, an accurate formalization of the property in temporal logic can pose a challenge. We have therefore experimented with an alternative method for specifying properties using a simple graphical user interface. Though this form of property specification is not as general, it covers many of the types of properties we are interested in. All properties specified in this way can be translated mechanically into temporal logic formulae, or also directly into property automata for use in the model checking process. Figure 5 shows the specification of the earlier property, checking for dialtone after an offhook.

The timeline editor allows the user to define events that are part of the required behavior on a horizontal line. Most events are markers that are used to identify the execution sequences of interest. These events

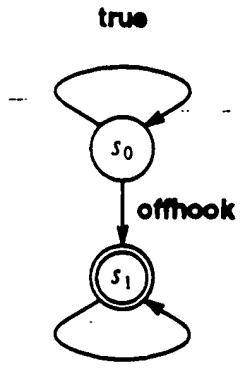

Fig. 4 — Property Automaton.
This ω-automaton (side box B) is automatically extracted from a temporal logic formula. It defines the set of behaviors that would violate the property, and is used in the verification process much like a 'pattern,' to search the set of all possible system executions for matches.

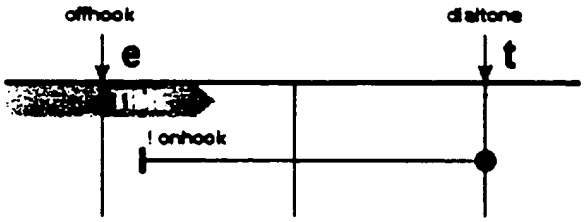

Fig. 5 — Timeline Editor
The timeline editor provides an intuitive alternative method for property specification. The user defines events and a test target, and mark multiple, possibly overlapping, intervals with constraints.

are labeled with an e in the diagram. Other events are required to appear in response to the earlier events. These *required* events are marked with an r (not shown in Figure 5). The last event of the sequence is always required and is marked with a t (for *target*). The model checker will flag an error if it can construct an execution sequence in which the markers (e) are present, but one or more of the required events (r and t) are missing.

The timeline editor also allows us to state that certain events must be absent for the execution to be of interest. In this case, this applies to onhook events. These conditions are specified as constraints on the execution, using a horizontal bar under the timeline to identify the precise part of the execution to which the constraint applies. For events or conditions that are not mentioned as events or in constraints, no restrictions apply.

The property definition shown in Figure 5 is automatically converted into the same automaton as shown in Figure 4, so the two methods of specification yield identical checks in this case.

4. Logic Model Checking

The formal models extracted from the source of the application are specified in the language of the LTL model checker SPIN [H97]. SPIN models define the behavior of systems of asynchronous processes that can communicate via message channels, rendezvous ports, or via shared data. SPIN converts the input specification into a product of automata. The global behavior defined by this product can be checked efficiently for a wide range of correctness properties using an automata theoretic model checking procedure [VW86]. To perform the check, SPIN starts by computing an automaton that captures all possible *violations* of a given correctness property. If the property is specified as a formula in linear temporal logic

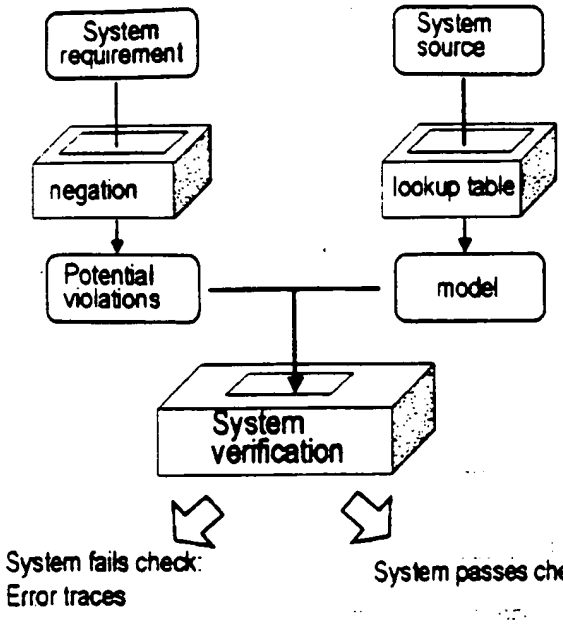

Fig. 6 — Overview of the Model Checking Process

[P77], for instance, SPIN takes the logical negation of the formula and converts it into a test automaton using the procedure defined in [GPVW95]. This automaton formalizes system executions that should not be feasible if the application is designed correctly. Next SPIN searches the intersection product of the language defined by the negated property automaton and the language defined by the automata that was extracted from the system. If the intersection product is empty, no violations of the property are possible and the system passes the test. If the intersection product is not empty it contains at least one complete execution that is both in the lanuage of the system (i.e., it is a possible execution of the system as specified) and in the language of the negated property (i.e., it constitutes the violation of a property). In this case SPIN will generates that execution sequence as proof that the property can be violated. In the FeaVer system the sequence is converted back into the source language of the application, using a reverse lookup in the table that was used to extract the formal model from the source of the application. The verification process is illustrated in Figure 6.

5. System Support: TrailBlazer

The main interface to the feature verification system is a standard web-browser. Through the browser the user can check on the verification status of all properties, lookup the text and the justification of each property, refer to the source text of the Telcordia feature requirement documents, and inspect reported error sequences in a number of different formats. An error sequence can be displayed as a message sequence chart in either ASCII or graphical form (Figure 7), or it can be displayed as a detailed dump of a series of concurrent execution traces interleaved in time, with one trace for each of the processes that participated in the failed execution. The detailed execution traces list all concrete C statements and conditions that are executed or evaluated during the execution, in time sequence. Typically such a sequence reveals subtle race conditions in the interleaving of actions that can lead to faults.

The main pieces of the infrastructure for the checking process: test drivers, the conversion lookup table, and the supporting text for properties are created and maintained with a standard text editor.

The source code of the application is maintained by the developers and parsed directly by the FeaVer software when a verification run is initiated.

Verification runs are always initiated by the user through the web interface. It would also be possible to automatically trigger a comprehensive series of checks each time that the FeaVer system detects that either the source of the application or the text of a property has changed, say in the early morning hours of every day. So far, however, we have not used this intrigueing possibility.

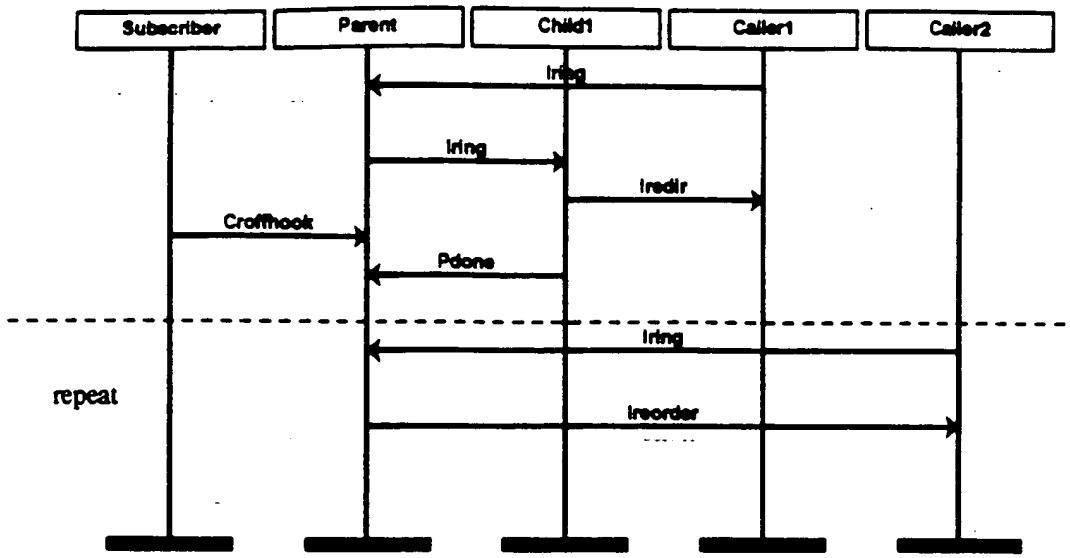

Fig. 7 — A Sample Property Violation

A sample execution sequence, shown here in graphical form as a message sequence chart, presented by the verification system as proof that executions are possible in which dialtone is not generated. In this case the property violation can occur if the subscriber has call forwarding, and happens to pick up the phone precisely when an incoming call is being forwarded. In an unlikely scenario, the call processing software can be made to delay the generation of dialtone arbitrarily long while the system is rejecting or forwarding more incoming calls. When the calls stop, the system will eventually timeout and deliver dialtone (not shown here). The scenario can also be presented as a trace of C statement executions.

To initiate a check, the user selects one or more properties and provisioning constraints from the web interface, and initiates the check with the click of a button. The remainder of this section describes what happens when the button is pushed. The tasks to be performed in the verification process are divided over a number of server applications that can run anywhere in the network. This capability to spread the work over several machines allows us to exploit large numbers of independent processors to assist in the execution of verification tasks. The additional processors are not necessary for FeaVer to perform its tasks, but, they can provide significant speedups. The network of processors that we have assembled for this purpose is called TrailBlazer (Figure 9). Four basic types of servers together provide the required functionality tb_prep, tb_sched, tb_exec, and tb_wrap, as illustrated in Figure 8 and explained in more detail below.

1. The FeaVer web browser sends a request to initiate one or more verification runs to a server called tb_prep. This server receives the property and provisioning information that the user provided and starts the process.

   It calls a program called pry to parse the C code of the application, identify the state routine, and convert it into an intermediate format, organized as state, event, transition triples. Another program, called catch then parses this intermediate format and generates a SPIN verification model, using the conversion map. Catch adds the user defined test drivers (defining context), suitably translated provisioning information, and the property, after converting it into automata form. On average there are two local states in the SPIN model for every line of source code in the application. The final model defines the behavior of 7 different types of processes (several of which are used to create multiple independent processing threads), 10 buffered message channels, and approximately 100 variables. The model is constructed in less than a second.

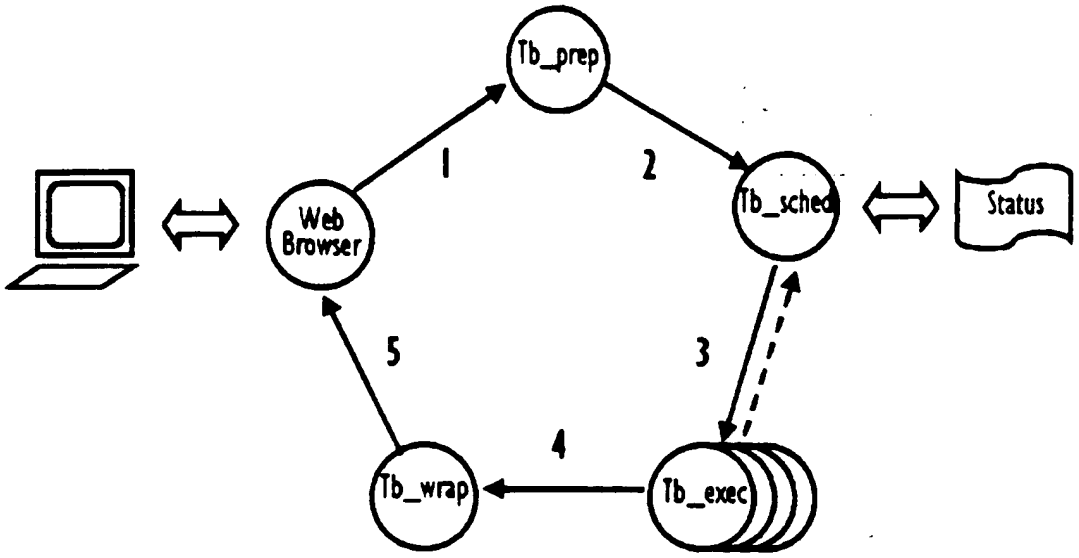

Fig. 8 — Servers and Workflow in the TrailBlazer System
When the user presses the 'check' button on the web browser, a sequence of steps is executed to mechanically verify the properties selected. Negative results of the verification typically flow back to the user within minutes after a check is initiated.

Tb_prep also generates a script that can be used to generate C code for a dedicated verifier for the model that was produced, and to compile and run that code. It now hands hands over the task to a central task scheduling server, tb_sched, by sending it a to the job file.

2. Tb_sched collects the information and adds it to its table of tasks to be completed. This server also collects offers to execute jobs from arbitrary workstations in our network. To make such an offer, the workstation runs a small server program called tb_exec. The volunteering workstation can run any type of operating system. The FeaVer servers, for instance, run under WindowsNT, and a number of dedicated PCs that act as compute-servers run under the Plan9 operating system [P95]. Fifteen PCs, shown in Figure 9, are permanently allocated to run verification jobs.

3. When the scheduler tb_sched identifies an available workstation it sends the corresponding server tb_exec a job script, with information on where any dependent information (e.g., files to be compiled) can be retrieved. The scheduler will attempt to have as many task performed in parallel as is possible, without overloading any one of the workstations. Typically no new job is assigned to a workstation until the previous one has completed. The search itself is performed with an iterative search procedure that optimizes our chances of finding errors quickly (side box A).

4. When a workstation completes a task it signals its renewed availability to the scheduler tb_sched and forwards the results of the run to the last server, on the FeaVer system, that performs postprocessing: tb_wrap.

5. Tb_wrap produces the ASCII and graphical format for error sequences, and generates detailed C traces. If no error was found, some statistics on the run are collected, to allow the user to judge the validity of the result (see **Avoiding false positives**). The statistics include the coverage of the property automaton, and the coverage of the model code as a whole. The information is entered in the database, and linked to the corresponding properties, so that it becomes accessible to the user via the web browser interface.

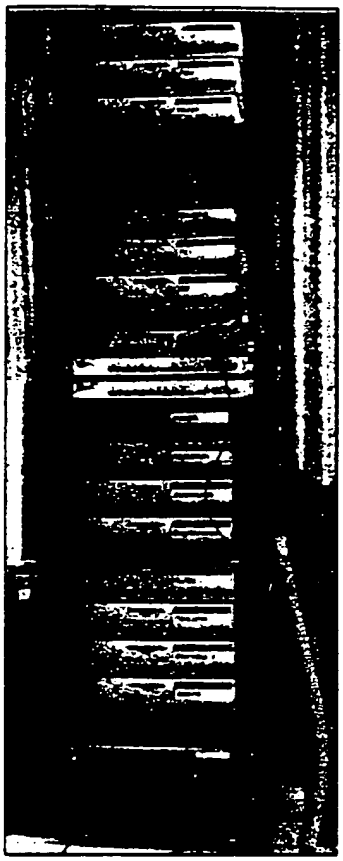

Fig. 9 — TrailBlazer Compute-Servers
Fifteen standard PCs, running the Plan9 operating system as compute servers, give the TrailBlazer *system a performance boost.*

Tracking Progress

When a comprehensive verification cycle is started for all properties that have been defined, for instance after an update of the source text of the application, it is of great interest to know immediately when an error sequence for a property has been discovered, so that it may be inspected. Typically this happens within the first few minutes of a comprehensive run, but it is of course not known in advance which properties might fail. The job scheduler tb_sched knows when the processing of an error sequence was completed, and can prompt the user, pointing at a URL where detailed information on the error sequence can be found, through the standard web browser. There is also a visual tracker, written in Tcl/Tk [O94], that shows the progress of the search with color bars, one for each property being verified. The bar turns red as soon as an error sequence has been discovered for the corresponding property. By clicking the bar, the detailed information on the sequence can be brought up in a web browser. This application is illustrated in Figure 10.

Separately, another small Tcl/Tk application can be used to track the actions of the scheduler: showing visually which machines have volunteered to execute jobs, which have been assigned a job and what the job details are, as illustrated in Figure 11.

Avoiding False Positives

From the point of view of the verification system, the best possible outcome of a verification attempt is the generation of an error sequence. There is a possibility, if the abstraction in the conversion table was chosen incorrectly, that the sequence is invalid and constitutes, what is called, a *false negative*. By inspecting the

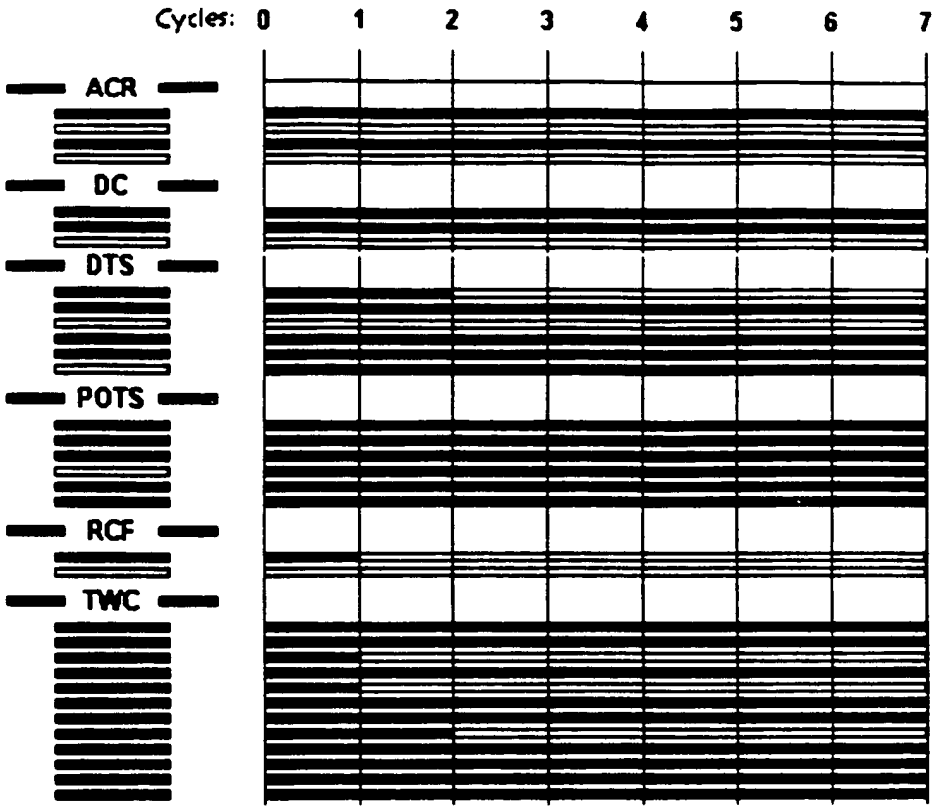

Fig. 10 — Property Verification Tracking

A list of all properties included in the current verification run is displayed. In the figure up to eight cycles in the iterative search refinement process are executed. The search stops, marking the progress line in red, as soon as an error is found.

sequence this can usually be determined quickly, and the abstraction can be adjusted to prevent reoccurrences. The absence of errors occurs when the application faithfully satisfies the property, but in this case it is possible that the property itself was in fact inadequate. This is called a vacuous property, cf. [KV99], or *false positive*, and it is addressed differently.

Consider a property of the type we discussed earlier

$$\Box(p \rightarrow X(\Diamond q))$$

It states that whenever a trigger condition p occurs then sometime thereafter, within a finite number of execution steps, a response q will follow, where q itself can either be a proper response or a discharge condition that voids the need for a response (e.g., an onhook event that voids the need for a dialtone signal). If all is well, there will be executions in which p occurs at least once. If there are no executions possible in which p occurs, then the formula is logically satisfied (note that the condition $\Box(p \rightarrow q)$ is satisfied when p is invariantly false). But even though the formula is strictly satisfied, it is almost certainly not what the user intended. The telltale sign of this false positive can be found in the number of states reached during the check for the automaton that corresponds to this property. In the case above, the property automaton never leaves its initial state. This occurrence can easily, and mechanically, be detected, so that the user can be warned to change the formulation of the property into a more meaningful one.

Because properties that do not generate error sequences can take the longest run times (i.e., they will pass through all iterative passes of the scheduler), the user can ask the scheduler to provide statistics on the runs that have been completed for a property. If the first few approximate runs for a property all leave the property automaton in its initial state, strong evidence that the property is void can be available within the first few minutes of the verification, and it is not necessary for the user to wait for the complete verification

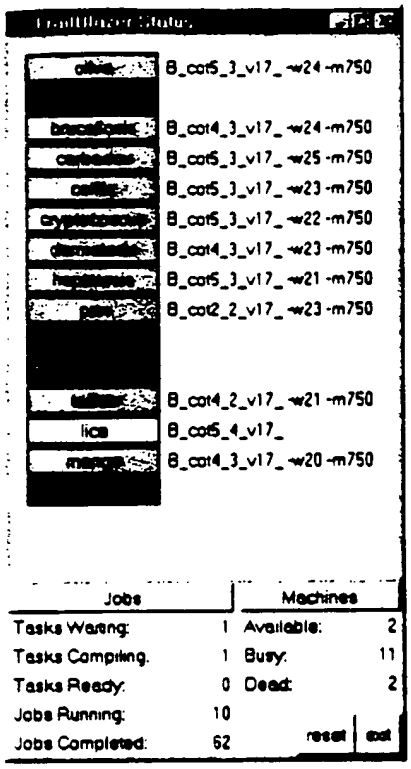

Fig. 11 — System Status Tracking

Optionally the user of the FeaVer *system can track the status of the workstations that have volunteered to run verification jobs. A color code identifies which of these workstations are currently free (green), which are busy (blue or yellow), which are dead (red). To the left of busy workstations is briefly indicated which job they are currently executing: a compilation (yellow) or a verification (blue).*

process to terminate.

Assessment

By concentrating on the central portion of the control code for call processing in PathStar, we were able to perform unusually thorough checks of critical system properties. This narrow focus, however, also pushed some interesting types of properties outside our reach. The main burden on the user of this checking process is the definition of meaningful properties, not on the mechanics of the checking process itself. The use of temporal logic can be a stumbling block, even for experienced users. To try to remedy this we developed the simple timeline editor, that is able to express the majority of the properties of interest in a fairly intuitive way. The vacuity check on apparently positive results from a verification run has also proved to be essential: it can be easy to state complex properties that are in retrospect meaningless. We built some mechanized checks to warn the user of such occurrences.

In a production environment, with strict project deadlines, the likelihood that a system error is addressed quickly is often inversely proportional to the amount of text that is needed to describe it. Execution sequences of thousands of steps that putatively violate complex logic properties are not likely to get quick attention. We therefore use the verification system in two phases: the first phase is used to identify all possible property violations, and the second phase is used to to generate the shortest possible example of each violation discovered, selecting the most likely manifestation of the error. This strategy has proven successful. In most cases an error can be demonstrated in no more than ten to fifteen steps, whereas an initial error sequence might contain hundreds and risk escaping notice.

6. Conclusions

At the time of writing, we have tracked the design, evolution, and maintenance of the PathStar call processing code over a period of approximately 18 months. In this period, the code grew fivefold in size and went through roughly 300 different versions, often changing daily. We intercepted approximately 75 errors in the implementation of the feature code by repeated verifications. Many of these errors were considered critically important by the programmers, especially in the early phases of the design. About 5 of the errors caught were also found independently by the normal system test group, especially in the later phases of the project. (The traditional testing, of course, addressed the PathStar system as a whole, and did not concentrate solely on the call processing code as we did.) In about 5 other cases the testers discovered an error that should have been within the domain of our verifications. These missed errors were caused by unstated or ambiguous system requirements; once the proper requirements were added into our database, the violations were caught.

Flaws can get enter into source code in the initial design stages, but also, and perhaps more frequently, during routine system maintenance. A portion of routine bug fixes will introduce new bugs into the code. The ability of the FeaVer system to repeat comprehensive verification runs immediately after bug fixes are made is therefore of great value. New incidences of property violations can be trapped instantly, while the rationale for a code change is still fresh in the mind of the developer.

In several cases we used our verification system in an unexpected way as a diagnostic tool. Occasionally the testers would run into a problem that could not be reproduced. By feeding the event sequence of such a test into the FeaVer system the error sequence could be reproduced in these cases and studied to determine which race conditions or event timings were responsible for its occurrence. In other cases the programmers of the system wanted to confirm their intuition about the occurrence or absence of certain conditions, such as a suspected unreachability of part of the code. The verification framework proved ideal to settle such question promptly.

The method of verification we have outlined in this paper should be generally applicable to distributed systems code written in most programming languages. Our aim in the coming years is to apply the method to a diverse set of applications, so that the checking process can be streamlined and made available for general use.

Acknowledgements

The authors are indebted to Ken Thompson and Phil Winterbottom, the principal authors of the call processing code in PathStar, for their insight and encouragement in this project. The authors are also grateful to Rob Pike and Jim McKie for their invaluable help with the setup of the compute-servers for the feature verification system.

7. References

[B92-96] LATA Switching Systems Generic Requirements (LSSGR), *FR-NWT-000064*, 1992 Edition. Feature requirements, including: *SPCS Capabilities and Features*, SR-504, Issue 1, March 1996. Telcordia/Bellcore.

[B99] Bozga, D.M., *Vérification symbolique pour les protocoles de communication.* PhD Thesis (in French), University of Grenoble, France, December 1999, (see Chapter 4).

[CAB98] Chan, W., Anderson, R.J., Beame, P., et al., Model checking large software specifications. *IEEE Trans. on Software Engineering*, Vol. 24, No. 7, pp. 498-519, July 1998.

[CGL94] Clarke, E.M., Grumberg, O., and Long, D.E., Model checking and abstraction. *ACM-TOPLAS*, Vol. 16, No. 5, pp. 1512-1542, September 1994.

[FSW98] Fossaceca, J.M., Sandoz, J.D., and Winterbottom, P. The PathStar™ access server: facilitating carrier-scale packet telephony. *Bell Labs Technical Journal*, Vol. 3, No. 4, pp. 86-102, Oct-Dec. 1998.

[GPVW95] Simple on-the-fly automatic verification of linear temporal logic. *Proc. Symposium on Protocol Specification, Testing, and Verification*, PSTV, Warsaw, Poland, pp. 3-18, 1995.

[H97] Holzmann, G.J., The model checker SPIN. *IEEE Trans. on Software Engineering*, Vol 23, No. 5, pp. 279-295, May 1997.

[HS99] Holzmann, G.J., and Smith, M.H. A practical method for the verification of event driven systems. *Proc. Int. Conf. on Software Engineering*, ICSE99, Los Angeles, pp. 597-608, May 1999.

[KK98] Keck, D.O., and Kuehn, P.J., The feature interaction problem in telecommunications systems: a survey. *IEEE Trans. on Software Engineering*, Vol 24, No. 10, pp. 779-796, October 1998.

[KV99] Kupferman, O., and Vardi, M.Y., Vacuity detection in temporal model checking, *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science, Springer-Verlag, 1999.

[K95] Kurshan, R.P., Homomorphic reduction of coordination analysis. *Mathematics and Applications*, IMA Series, Springer-Verlag, Vol. 73, pp. 105-147, 1995.

[AL91] Abadi, M., Lamport, L., The existence of refinement mappings. *Theoretical Computer Science*, Vol. 82, No. 2, May 1991, pp. 253-284.

[P95] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., Winterbottom, P., Plan 9 from Bell Labs, *Computing Systems*, Vol. 8, No. 3, pp. 221-254, and Vol. 2, No. 2, pp. 133-153, 1995.

[P77] Pnueli, A., The temporal logic of programs. *Proc. 18th IEEE Symposium on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.

[S98] Schneider, F., Easterbrook, S.M., Callahan, J.R., and Holzmann, G.J., Validating Requirements for Fault Tolerant Systems using Model Checking, *Proc. International Conference on Requirements Engineering*, ICRE, IEEE, Colorado Springs Co. USA, pp. 4-14, April 1998.

[O94] Ousterhout, J.K., *Tcl and the Tk Toolkit*. Addison-Wesley Publ. Co., Reading, Ma., 1994.

[S65] Strachey, C., An impossible program. *Computer Journal*, Vol. 7, No. 4, p. 313, January 1965.

[T90] Thomas, W., Automata on infinite objects. *Handbook on Theoretical Computer Science*, Volume B, Elsevier Science, 1990.

[T36] Turing, A.M., On computable numbers, with an application to the Entscheidungs problem. *Proc. London Mathematical Soc.*, Ser. 2-42, pp. 230-265 (see p. 247), 1936.

[VW86] Vardi, M.Y., and Wolper, P., An automata-theoretic approach to automatic program verification. *Proc. Symp. on Logic in Computer Science*, pp. 322-331., Cambridge, June 1986.

Sid Box A: Iterativ Search Procedure

SPIN provides support for performing verification jobs either exactly or with varying degrees of precision, using proof approximation techniques. The benefit of an exact run will be clear. An exact verification, however, can be time consuming for larger problems. An approximate answer that can be delivered quickly is often of more value to a user than a precise answer that takes much longer to compute. In an approximate verification both the quality and the speed of the run can be controlled with a parameter. As the thoroughness of the run increases or decreases, so do the time requirements. An approximate verification in effect performs a random sampling of the behavior of a system, in an effort to find violations of system requirements. With this technique we can find a practical compromise between verification and testing. The coverage that is achieved in even approximate verification runs is significantly larger than what is achievable with traditional testing techniques.

Our experience is that when property violations are possible, even very approximate verification runs can identify them. The verification system uses this fact to enforce an iterative search refinement method for each verification task. The iterative search procedure starts by allocating the fastest possible, and most approximate, runs for each task. The first of these runs typically completes in under a second of CPU time. If the run finds an error, the remainder of the runs can be abandoned, and the error sequence can be processed for inclusion in the FeaVer database. If no error is found, the coverage is increased. The second run may take two seconds, proceeding to four, eight, sixteen second runs, and so on until either an error is found or maximal coverage was reached (and with that proof that no violations of the corresponding property are possible).

With, say, 200 verification runs to be performed and 20 workstations available to perform the runs, the first approximate runs can be completed in about ten seconds for all jobs combined. In each new iteration all jobs that produced errors are deleted from the workset, and the scan becomes more thorough for the remaining properties. With this procedure it typically takes a few minutes to identify the first property violation in a large set of verification tasks. After about five minutes a representative selection of violations is normally available, with the gaps filled in in subsequent searches. The iterative search procedure is abandoned after about an hour, whether it has reached fully exact results or not. The rationale is that within an hour one normally will have looked at the property violations and formulated corrections of the source code. Further error sequences would be of little use, since the source code has by now changed, and more value can be derived from a new scan of all properties for the new version of the source.

Once all verification tasks have been completed, an optional second phase of the verification is performed by reinspecting every error sequence found and, again iteratively, searching for a shorter equivalent (see also Assessment).

Side Box B: Omega Automata

The formal definition of an ω-automaton, as shown in Figure 4, differs slightly from that of a standard finite automaton. Instead of accepting (input) sequences of finite length, like a standard finite automaton, an ω-automaton accepts only sequences (in our case representing system executions) of infinite length. There are several ways to define the acceptance conditions for an ω-automaton [T90]. The definition used in SPIN is known as Büchi acceptance. It states that a sequence is accepted if and only if it visits at least one accepting state in the automaton (indicated with a double circle in Figure 4) infinitely often.

The automaton in Figure 4 is also non-deterministic, which makes it's behavior less obvious. In the model checking process, the transitions of this automaton are 'matched' one by one against the execution steps of the system. Execution starts with the property automaton in its initial state s_0. After each step of the system the property automaton is forced to make a transition. To do so it can choose only from transitions with a label that evaluates to *true* at this point in the execution. The self-loop on s_0 in Figure 4 can always be traversed, since it's label necesarily evaluates to *true*. The transition from s_0 to s_1 can only be taken when an offhook is detected. Note carefully that *if* an offhook is detected the property automaton can either stay in s_0, and ignore this event, or move to s_1 and start the wait for a *dialtone* (i.e., it makes a non-deterministic choice). The verifier will check the consequences of either choice. The latter is important because we want to make sure that every occurrence of an offhook is followed by a dialtone, not just the first occurrence.

Once the property automaton reaches state s_1 it can only remain there in the absence of dialtones and

onhooks. A sequence is formally *accepted* by the automaton if and only if it is possible for the property automaton to remain in s_1 forever, infinitely often traversing the self-loop on that state. If a dialtone is detected within a finite number of steps, the attempt to match the corresponding execution to the property automaton fails, which means that the execution satisfies the original requirement and does not constitute a violation. The matching process stops at this point. The model checker will abandon the search of this execution and explore other possible executions instead, in a hunt for possible violations. There need not be a back-edge from state s_1 to state s_0, since all behaviors of interest are already captured by the nondeterminism on s_0 (see above).